Polish Parsers, Step by Step

R. John M. Hughes
Computing Science
Chalmers University of Technology
SE-412 96 Gøteborg, Sweden
rjmh@cs.chalmers.se

S. Doaitse Swierstra
Institute of Information and Computing Sciences
Utrecht University P.O. Box 80.089
3508 TB Utrecht, the Netherlands
doaitse@cs.uu.nl

ABSTRACT

We present the derivation of a space efficient parser combinator library: the constructed parsers do not keep unnecessary references to the input, produce online results and efficiently handle ambiguous grammars. The underlying techniques can be applied in many contexts where traditionally backtracking is used.

We present two data types, one for keeping track of the progress of the search process, and one for representing the final result in a linear way. Once these data types are combined into a single type, we can perform a breadth-first search, while returning parts of the result as early as possible.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Algorithms

Keywords

parser combinators, breadth-first search, Polish representation, ambiguous grammars, online results, GLR parsing

1. INTRODUCTION

A basic parser combinator library can be formulated in a few lines (see figure 1, [1]). We use the interface described in [4]:

- p `<*>` q parses p followed by q, where p returns a function that is applied to the result of q in order to construct the result of the sequential composition
- p `<|>` q parses either p or q
- *pSucceed* v is a parser that immediately succeeds without accepting any input and returns v
- *pFail* is the unit of `<|>`
- *pSym* s is the parser that recognizes the symbol s and returns this s
- f `<$>` p applies f to the result of parsing p,
- p `<*` q parses a p and recognizes a q and returns just the result of p, and
- c `<$` p parses a p but just returns c.

Note that this is not a monadic interface.

Such simple libraries thus far always have had some shortcomings:

1. Backtracking implementations based on the list of successes method keep a reference to the full sequence of input tokens, until it has become clear that no further alternatives will be found.
2. Parsers only start producing a result once the complete input has been examined, and thus do not exhibit *online* behaviour.
3. Simple backtracking implementations are grossly inefficient when dealing with ambiguous grammars; for each possible parse for a part of the text, the rest of the text is parsed once. This becomes immediately clear if we look at the code for the sequential composition of two parsers: p `<*>` q. If the parser p can succeed in several different ways by consuming the same prefix of the input, the parser q is called once for each of those successful parses of p, with the same remaining input as argument.

As a consequence of the first two points the constructed result and the initial input are both present in memory once a complete parse has been found. This is highly undesirable if we process input that represents a long list of similar items such as a *bibtex* file, or when we describe a lexical scanner that returns a list of input tokens. In such situations we do not want first read and recognize a complete input file before producing any output: output should be produced as soon as an individual element has been recognized.

If the grammar is $LL(1)$, as required by the parsers described by Swierstra and Duponcheel [6], this problem does not occur, since it is a property of the grammar that once a successful alternative is taken, backtracking will not find other possible parses and can thus be avoided; as consequence references to the input have not to be preserved and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'03, August 25–29, 2003, Uppsala, Sweden.
Copyright 2003 ACM 1-58113-756-7/03/0008 ...$5.00.

```
infixl 3 <|>
infixl 4 <*>, <*, <$, <$>

type Parser a = String → [(a, String)]
(<|>)     :: Parser       a   → Parser a  → Parser a
(<*>)     :: Parser (b →  a)  → Parser b  → Parser a
pSucceed  ::                  a             → Parser a
pFail     :: Parser a
pSym      :: Char                           → Parser a

pSucceed  v  inp  =  [(v, inp)]
pFail             =  []
(p <|> q)    inp  =  p inp ++ q inp
(p <*> q)    inp  =  [(f a, rr)   | (f, r)  ← p inp
                                  , (a, rr) ← q r
                     ]
pSym  a      inp  =  case inp of
                       (s : ss) → if     a ≡ s
                                  then [(s, ss)]
                                  else []
                       []       → []

f  <$>  q  = pSucceed f  <*>  q
p  <*   q  = const <$> p  <*>  q
f  <$   q  = const f      <$>  q
```

Figure 1: Backtracking implementation

the constructed result can be returned as it is being produced. For non $LL(1)$ grammars the problem can be partially cured by assuming that by default no backtracking has to take place, and to require explicit annotation in the code where a longer look-ahead is needed (the *try* construct in the Parsec library [3])[1]. This works well for grammars that are (almost) in the $LL(1)$ class, provided the programmer of the parser indicates explicitly at which points the $LL(1)$ assumption does not hold. As with placing *cut* clauses in a Prolog program, this requires a grammar analysis from the side of the user of the library. Sometimes this is easily done, but it can also be a complicated and error prone process, and is in any case something the writer of the parser has to be continuously aware of.

In an earlier paper ([4]) we have explained how we can implement a parsing algorithm that performs a breadth-first instead of a depth-first search for a complete parse, and sequentially accesses the input stream; it furthermore removes the $LL(1)$ restrictions and works with unbounded look-ahead. This completely cures the first problem mentioned above, without any annotations provided by the programmer nor a need to understand how the parsing process proceeds internally, but it introduces the second problem mentioned.

In this paper we show how to cure the second and third problem mentioned: the build-up of a complete result in memory and the inefficient handling of ambiguous grammars. The only difference that remains from very general parsing processes for context-free languages such as GLR ([7]) is that, basically using a top-down parsing method, we cannot handle left recursive grammars and left-factorization may be needed in certain situations to improve performance. We do not see this as a problem at all, since in practice uses of left recursion usually can be nicely captured using special combinators for recognizing list like structures.

The key idea in this paper is a new data type that enables us to represent any value in a linear way (i.e. as a sequence of fragments), thus enabling us to intersperse the result value with information about progress of the parsing process. This type is introduced in section 2. In section 3 we start by repeating the way we construct breadth-first recognizers, and subsequently extend them with an incremental construction of the parsing result. In section 4 we show that the advantages that are provided by monadic parser interfaces are not lost, and that it is still possible to parameterize parsers with values stemming from earlier recognized parts of the input. In section 5 we extend the data type that we have introduced so it becomes possible to parse ambiguous grammars efficiently, and in section 6 we discuss some optimizations to the code given, that were left out for the sake of presentation. In section 7 we discuss some practical aspects of a library built on the ideas presented in this paper, whereas in 8 we reflect on what we have done and mention some further applications of the ideas presented.

2. THE POLISH REPRESENTATION

While parsing we will recognize the fragments of the value that witnesses a complete and successful parse in a sequential way, so for a moment we forget about parsing and focus on the question whether we can find a data type that is suitable for representing such a linearly fragmented value. In order to answer this question we let ourselves be inspired by the way values are represented in Polish expressions.

2.1 The Problem

Polish notation, in which we write operators before their operands, is a way of writing expressions unambiguously in a linear way, i.e. without brackets. For example, $1+2\times 3$ is written as $+\ 1\times 2\ 3$, and $\sin x \times \pi/4$ is written $\times \sin x\ /\pi\ 4$. One can interpret Polish by inspecting the operator at the beginning of the expression, evaluating the correct number of arguments, and then applying the operator to the argument values.

To do so correctly, one must know the arity of each operator, which could be problematic if we allow operators to be arbitrary functions. But we can apply the usual trick of currying all functions, and taking *function application* to be the only operator, with functions thus becoming ordinary values. Writing function application as '@', $1 + 2$ is now written as $@\ @ + 1\ 2$.

The problem we consider in this paper is the design of a *fully typed* representation and interpreter for Polish expressions in Haskell. The essence of a Polish representation is its linear nature, but unfortunately we cannot use ordinary lists since we require the elements to be of different types.

More specifically, we are looking for a data type with constructors *App* and *Val* such that we can represent the expression above by:

$$App\ (App\ (Val\ (+))\ (Val\ 1\ (Val\ 2\ \bot)))$$

Since we know from the syntax of Polish expressions when an expression is complete the $\bot$ at the "end of the list" is unimportant. We will from now on write such lists as functions,

[1]Unfortunately parsers constructed by the Parsec library do not exhibit online behaviour due to their monadic formulation.

so that we can use function composition for concatenation, and write instead for the list above:

$$App \cdot App \cdot Val\ (+) \cdot Val\ 1 \cdot Val\ 2$$

We will aim to write an interpreter such that

$$interpret\ (App \cdot App \cdot\ Val\ (+) \cdot\ Val\ 1 \cdot Val\ 2) \equiv 3$$

This problem is interesting because Polish expressions must be able to contain values of many different Haskell types. In fact, we do not believe the Haskell 98 type system is powerful enough to allow this, but with the extensions in the current versions of Hugs and GHC, it is. The reader may like to try to solve the problem unaided, before reading further: it is quite a challenge!

2.2 A Simpler Problem

If we cannot immediately see how to represent Polish expressions in Haskell, we can consider the simpler problem of representing ordinary expressions. We may try to define a type *Expr* with a binary constructor *App* and a unary constructor *Val*, so that 1+2 is represented by $App\ (App\ (Val\ (+))\ (Val\ 1))\ (Val\ 2)$, for example.

Of course, the type of our representation should guarantee that no type errors arise during expression evaluation. To ensure that we only represent well-typed expressions, we had better make the type that an expression evaluates to into a part of its own type. That is, we define a type $Expr\ a$ representing expressions that evaluate to a value of type a. Now it is clear what the types of the constructors should be:

$$\begin{array}{l} Val\ \ ::\ a\ \rightarrow Expr\ a \\ App\ ::\ Expr\ (b \rightarrow a) \rightarrow Expr\ b \rightarrow Expr\ a \end{array}$$

This suggests the type definition

$$\begin{array}{lcl} \textbf{data}\ Expr\ a & = & Val\ a \\ & \mid & App\ (Expr\ (b \rightarrow a))\ (Expr\ b) \end{array}$$

but here the variable b is unbound. Fortunately, current Haskell extensions permit us to quantify type variables existentially in type definitions: we can say that an *App* expression contains an $Expr\ (b \rightarrow a)$ and an $Expr\ b$ for *some* type b, but this type may vary from *App* node to *App* node. Thus we write the type definition as[2]

$$\begin{array}{lcl} \textbf{data}\ Expr\ a & = & Val\ a \\ & \mid & \exists b.\ App\ (Expr\ (b \rightarrow a))\ (Expr\ b) \end{array}$$

which introduces the constructors with exactly the types we want.

An evaluator for such expressions is easy to write:

$$\begin{array}{lcl} eval & :: & Expr\ a \rightarrow a \\ eval\ (Val\ a) & = & a \\ eval\ (App\ e1\ e2) & = & \textbf{let}\ f = eval\ e1 \\ & & \quad\ a = eval\ e2 \\ & & \textbf{in}\ f\ a \end{array}$$

We remark only that this definition uses polymorphic recursion, which is allowed in Haskell provided that a type signature is given explicitly.

[2]The keyword for this construct used to be **forall**, but GHC nowadays also accepts **exists**.

2.3 The Polish Type

Returning to Polish expressions, it is tempting by analogy to define a type $Polish\ a$, representing Polish expressions of type a, but a moment's reflection shows this will not be enough. In the expression $App \cdot Val\ (const\ 1) \cdot Val\ 2$, for example[3], the Polish subexpression $Val\ (const\ 1) \cdot Val\ 2$ *after* the initial *App* represents *two* values, namely *const* 1 and 2, of different types. In general, a Polish expression may represent arbitrarily many values, and therefore we cannot expect it to be possible to give the type *Polish* just a single type parameter.

Our solution is to give the *Polish* type an additional parameter, representing the type of "the part at the right hand side of a $\cdot$". Thus the expression above will have the type $s \rightarrow Polish\ Int\ s$, while the subexpression $Val\ (const\ 1) \cdot Val\ 2$ will have the type $s \rightarrow Polish\ (b \rightarrow Int)\ (Polish\ Int\ s)$. We need to assign the constructors the following types:

$$\begin{array}{lll} Val\ \ ::\ a \rightarrow Polish\ b\ s & & \rightarrow Polish\ a\ (Polish\ b\ s) \\ App\ ::\ Polish\ (b \rightarrow a)\ (Polish\ b\ s) & & \rightarrow Polish\ a\ s \end{array}$$

Notice that these types account for the number of values the expression represents, which is increased by *Val* and decreased by *App*.

Once again, we can handle the type variable b in the type of *App*, which does not appear in the type of its result, by an existential quantification. But the type of *Val* presents a different problem: its result type is not of the form $Polish\ a\ s$, which means that we cannot directly give *Val* this type via a data type definition. Fortunately, the natural type above is an instance of

$$Val :: a \rightarrow s \rightarrow Polish\ a\ s$$

which does have a result type of the right form, so we will simply take this to be the type of *Val* instead, and define the following *nested* data type:

$$\begin{array}{lcl} \textbf{data}\ Polish\ a\ s & & \\ = & & Val\ a\ s \\ \mid & & \exists b.\ App\ (Polish\ (b \rightarrow a)\ (Polish\ b\ s)) \end{array}$$

Now indeed, terms of the sort we considered in the introduction are well-typed. For example,

$$\begin{array}{l} App \cdot App \cdot Val\ (+) \cdot Val\ 1 \cdot Val\ 2 \\ \quad :: s \rightarrow Polish\ Int\ s \end{array}$$

2.4 A Polish Interpreter

Our second goal is an interpreter for Polish expressions, that is, a function

$$interpret :: (s \rightarrow Polish\ a\ s) \rightarrow a$$

But this type does not take into account that some Polish expressions represent *many* values – for example, the Polish expression following an *App*. We therefore generalize to a function *eval*, which returns both the value of the first complete sub-expression, and the unconsumed part of the Polish input:

$$eval :: Polish\ a\ s \rightarrow (a, s)$$

We can then define

$$interpret\ v = (fst \cdot eval)\ (v\ ())$$

[3]Where *const* x is a constant function returning x

The *eval* function is now straightforward to define:

$$\begin{array}{lll} eval & :: Polish\ a\ s \to (a, s) \\ eval\ (Val\ a\ s) & = & (a, s) \\ eval\ (App\ s) & = & \textbf{let}\ (f,\ s') = eval\ s \\ & & \quad\ (a,\ s'') = eval\ s' \\ & & \textbf{in}\ (f\ a, s'') \end{array}$$

Once again, polymorphic recursion is needed to type this definition, so the type signature cannot be omitted. Since Haskell's **let**-expression binds lazily, s' will not be evaluated if f is a lazy function – for example:

$$interpret\ (App \cdot Val\ (const\ 1) \cdot Val\ \bot) \equiv 1$$

This is exactly what we are looking for in an online parser, since it means that we can start evaluating a *Polish* expression, and even construct a part of its result, before the entire expression has become available. It will be this property that facilitates the online behaviour of the parser combinator library that we will introduce now.

3. PARSING POLISH

3.1 Motivation

As we mentioned in the introduction, the design of the Polish type arose as the result of an attempt to define a parser combinator library that constructs parsers that return their result in an online way; that is, as soon as it can be unambiguously decided that some value will be used to construct the final result it becomes available to the callee of the parser, just as a *foldr* may already return part of its result when processing a list of elements.

We also want to avoid a problem in many parsing libraries related to choice. Every parser must be able to choose between alternative possible parses for the same input – for example, between parsing an addition and parsing a multiplication. Normally one parse fails quickly, since they accept quite different constructs. But in backtracking parsers this choice operation results in a space leak. The problem occurs when the first alternative to be tried actually succeeds, and therefore consumes a large amount of input, but the second alternative is retained as a backup. To retain the possibility of constructing the second parse, the entire input must be kept available – even though the second parse in all probability would fail quickly were it to be tried. Ever since Wadler [8], backtracking parsing libraries have included more or less ad hoc solutions to this problem, permitting the second parse to be discarded when the first one 'commits' to its result. These solutions unfortunately lead to unmodular code: when we write the first alternative, we must decide when to commit based on our knowledge of all its alternatives. When the grammar is later modified, such parsers have a tendency to stop working.

One of the consequences of taking the backtracking (or depth-first) approach is that it is only after the parsing process has been completed that the result becomes available; the information about how the result was reached precedes the actual result. One moment of reflection however tells us that from a parsing point of view there is nothing which prevents us from already returning that part of the result for which no parser "threads" are competing anymore. So as soon as we are no longer interested in look-ahead information we can produce the result recognized thus far. This will usually be after inspecting only a few input tokens.

$$\begin{array}{lll} rSucceed & = & R\ (\lambda k\ input \to k\ input) \\ rFail & = & R\ (\lambda k\ input \to Fail) \\ R\ p\ \texttt{<*>}\ R\ q & = & R\ (\lambda k\ input \to p\ (q\ k)\ input) \\ R\ p\ \texttt{<|>}\ R\ q & = & R\ (\lambda k\ input \to p\ k\ input \\ & & \qquad\qquad\qquad\quad `best` \\ & & \qquad\qquad\qquad\quad q\ k\ input \\ & &) \\ pSym\ a & = & R\ (\lambda k\ input \to \\ & & \qquad \textbf{case}\ input\ \textbf{of} \\ & & \qquad (s : ss) \to \textbf{if}\ \ a \equiv s \\ & & \qquad\qquad\qquad\ \textbf{then}\ Shift\ (k\ ss) \\ & & \qquad\qquad\qquad\ \textbf{else}\ Fail \\ & & \qquad [\,] \to Fail \\ & &) \end{array}$$

$$\begin{array}{llll} Fail & `best` p & = p \\ q & `best` Fail & = q \\ Done & `best` Done & = error\ \texttt{"ambiguous grammar"} \\ Done & `best` q & = Done \\ p & `best` Done & = Done \\ (Shift\ v) & `best` (Shift\ w) & = Shift\ (v\ `best`\ w) \end{array}$$

$$recognize\ (R\ r)\ input = r\ Done\ input$$

Figure 2: The recognizing combinators

The question that arises now is how to *combine the two useful properties*: not unnecessarily retaining references to the input, and returning parts of the result as soon as possible.

The basic idea is that the parser returns a interleaving of two kinds of sequences: a sequence of fragments constituting the result we are interested in, and a trace of the parsing process itself that is used for the breadth-first search.

3.2 Recognisers

Let us for a moment postpone the question of how a parser constructs a result, and focus on recognizers instead, where a *recognizer* is a degenerate parser that does not build up value. Our recognizers will just indicate their progress, and finally their success or failure by returning a value of type:

$$\begin{array}{lll} \textbf{data}\ Progress & = & Shift\ Progress \\ & | & Done \\ & | & Fail \end{array}$$

in which *Shift* represents the successful recognition of a single input token, *Fail* indicates a dead end, and *Done* signals the completion of a successful parse.

A recognizer take two arguments: a continuation for recognizing the rest of the input once once this recognizer has succeeded, and the input of which a prefix has to be recognized. For the sake of presentation we will assume that the input is a sequence of characters:

$$\begin{array}{lll} \textbf{newtype}\ Rec = R & & (String \to Progress) \\ & \to & (String \to Progress) \end{array}$$

We can now define the combinators for sequential composition <*>, alternative composition <|>, the basic always successful recognizer *rSucceed*, the always failing recognizer *rFail*, and a function for constructing a single-symbol recognizer *rSym*. They are all given in figure 2.

The choice between two alternatives is implemented by

the function *best* (figure 2), that traverses the two alternatives in a synchronized way: when both of its alternatives present a *Shift*, the function produces one while removing the *Shift*'s at the same time from both alternatives. If either of its alternatives presents a *Done* this indicates we have found a successful parse. In case both alternatives present *Done* at the same time we have found more than one successful parse, and thus have been parsing with an ambiguous grammar. Notice that a failing alternative can be discarded altogether as soon as it presents its *Fail* step, so that a recognizer built using `<|>` will ultimately return either one of its branches, provided the grammar is not ambiguous. Note however that the function *best* is looking only as far as needed into the progress information in order to decide which alternative to choose; thus the strategy we follow deals with unbounded look-ahead, and only looks ahead when needed.

3.3 Parsers

The next step is to extend the recognizers to real parsers, that return a parsing result. The essence of this paper is that for the result type of a parser we define a data type *Steps*, that combines *both* the *Polish* and *Progress* data types (figure 3); our parsers deliver information *both* regarding their progress, just like our recognizers, *and* their result, expressed in *Polish*. If we think of the *Progress* type as representing a sequence of recognition steps, and the *Polish* type as a sequence of applications and values, then the *Steps* type represents an *interleaving* of these two sequences. Because of the freedom we will use in changing an interleaving it is no longer guaranteed that the *Done* constructor marks the end of a sequence. So we have slightly changed its introduction: it can now contain further steps. In the function *parse* (figure 3) we use this facility to pass back the unused part of the input in the result.

The reason for linearising the output of a parser in Polish, is that it can be interleaved with the recognition trace. So our first attempt for the type representing the new parsers is:

```
newtype Par a
  = P (∀w. (String →          w) →
           (String → Steps a  w)
  )
unP (P p) = p
```

In figure 3 we also define an evaluation function *evalSteps* for *Steps* which just discards the recognition trace and otherwise behaves like *eval*.

The definition of a small library of basic parsing combinators appears in figure 4 (apart from the new function *best*, to which we return below).

Using these definitions we may construct a parser for an addition expression, using a very simple parser for integers:

```
pPlus = (+) <$> pInt <* pSym '+' <*> pInt
pInt  =       0 <$ pSym '0'
        <|> ...
        <|> 9 <$ pSym '9'
```

```
data Steps      a s
  =      Val    a s
  | ∃b.  App    (Steps (b → a) (Steps b s))
  |      Shift  (Steps a s)
  |      Done   (Steps a s)
  |      Fail

evalSteps            :: Steps a s → (a, s)
evalSteps (Val a  s) = (a, s)
evalSteps (App    s) = let (f, s')  = evalSteps s
                           (a, s'') = evalSteps s'
                       in  (f a, s'')
evalSteps (Shift  v) = evalSteps v
evalSteps (Done   v) = evalSteps v
evalSteps Fail       = error "wrong input"
```

Figure 3: The data type *Steps*

```
pSucceed :: a      → Par a
pFail    ::        Par a
pSym     :: Char   → Par Char

pSucceed a   = P (Val a·)
pFail        = P (λ_ _ → Fail)
P p <*> P q  = P ((App·) · p · q)
P p <|> P q  = P (λk input
                   → (p k input) `best` (q k input)
                 )
pSym a
  = P (λk input →
         case input of
           (s : ss) → if   a ≡ s
                      then (Shift · Val s · k) ss
                      else Fail
           []       → Fail
      )
parse p input
  = let (result, rest) =
          evalSteps (p (λrest → Done (Val rest ()))
                       input
                    )
    in  (result, fst · evalSteps $ rest)
```

Figure 4: The parsing combinators

The result of parsing $1 + 2$ with this parser is:

$$\begin{array}{llllllll} App \cdot & App \cdot & App & \cdot & Val\ \ const & \cdot & & \\ & & & & App \cdot\ Val\ (+) & \cdot & & \\ & & & & \quad Shift & \cdot App \cdot & Val\ (const\ 1) & \cdot \\ & & & & & & Val\ \texttt{'1'} & \cdot \\ & & Shift & \cdot & Val\ \texttt{'+'} & \cdot & & \\ & Shift \cdot App & & \cdot & Val\ (const\ 2) & \cdot & & \\ & & & & Val\ \texttt{'2'} & & & \end{array}$$

$$Done$$

which, when we evaluate it, yields $const\ ((+)\ (const\ 1\ \texttt{'1'}))\ \texttt{'+'}\ (const\ 2\ \texttt{'2'})$, or (after simplification) $1 + 2$, that is, 3.

Now suppose that we try to parse the same input, using an analogous parser for multiplications instead. This results in a sequence of steps terminated by *Fail*:

$$\begin{array}{llllllll} App \cdot App \cdot & App & \cdot & Val\ \ const & \cdot & & & \\ & & & App \cdot\ Val\ (*) & \cdot & & & \\ & & & \quad Shift & \cdot App \cdot & Val\ (const\ 1) & \cdot \\ & & & & & Val\ \texttt{'1'} & \cdot \\ & Fail & & & & & & \end{array}$$

Now, a parser which parses *either* an addition or a multiplication must choose between these parses, and we can see immediately that this is more complicated than in the case of recognizers. The choice must be made based on the progress information embedded in the sequence, and indeed cannot be made until the second *Shift* has been produced, but this information is preceded in each parse by Polish operations for creating the parse result. Moreover, since the parse result differs between the two parses, we clearly cannot start returning it until we know which parse will eventually be chosen.

Fortunately, we can freely change the interleaving of the Polish operations and the recognition trace as long as we do not reorder either subsequence, since they are essentially independent of each other. In particular, we can move the first progress step to the front of a sequence, thus making it possible to make a progress decision. We therefore define a function *getProgress* that moves the first element of the embedded progress trace to the head of the interleaved sequence (or formulated differently, pushes the value constructing elements behind the first progress step), returning a parse with a top constructor that is either *Shift*, *Fail* or *Done*. We define the *best* function on parsers to convert its arguments into this form, if necessary, and define some new alternatives for the function *best*.

$$\begin{array}{llllll} Fail & `best` & p & = & p & \\ q & `best` & Fail & = & q & \\ Done\ \_ & `best` & Done\ \_ & = & error\ \texttt{"ambiguous grammar"} & \\ Done\ a & `best` & q & = & Done\ a & \\ p & `best` & Done\ a & = & Done\ a & \\ Shift\ v & `best` & Shift\ w & = & Shift\ (v\ `best`\ w) & \\ p & `best` & q & = & getProgress\ id\ p & \\ & & & & \quad `best` & \\ & & & & getProgress\ id\ q & \end{array}$$

This definition is closely analogous to the definition for recognizers.

The *getProgress* function traverses its input searching for a step, accumulating the step-free part of its input in its first parameter as a function, and then reinserts the accumulated information after the first progress step (we will make sure that every parse ends with a *Done* step):

$$\begin{array}{llll} getProgress\ f\ (Val\ a\ \ s) & = getProgress & (f \cdot Val\ \ a\ ) & s \\ getProgress\ f\ (App\ \ \ s) & = getProgress & (f \cdot App\ \ \ ) & s \\ getProgress\ f\ (Done\ \ p) & = Done & (f\ p) & \\ getProgress\ f\ (Shift\ \ s) & = Shift & (f\ s) & \\ getProgress\ f\ (Fail\ ) & = Fail & & \end{array}$$

Unfortunately, this straightforward definition of *getProgress* does not type-check! The reason is that, in the first equation, we apply *getProgress* to s of type s – a type variable. The problem is that the second component of a *Val need not be of Steps type*, so we cannot apply *getProgress* to it.

Our solution is to *overload getProgress* instead; so we declare

```
class HasProgress st where
  getProgress :: (st → Steps x y) → st → Steps x y
```

and make the above definition the instance at type *Steps*:

```
instance HasProgress s ⇒ HasProgress (Steps a s) where
  ... as before...
```

This instance declaration defines *HasProgress* (*Steps a s*) in terms of *HasProgress s*; we also need a 'base case', which we can take to be:

```
instance HasProgress () where
  getProgress f () = Fail
```

Provided we see to it that top-level parsers produce a *Steps a* () for some a, we will now be able to apply *getProgress* to all intermediate parses.

The introduction of this class now enforces a slight adaptation of our parser type:

```
newtype Par a
  = P  (∀w. (HasProgress w ⇒
              (String → w) → String → Steps a w)
       )
```

With this definition, the first version of our simple Polish parsing library is complete, and can be used to write online parsers which do not leak space.

4. MONADIC INTERFACE

One of the advantages of a monadic interface ([2]) over the interface chosen here is that one can parameterize parsers on the results of earlier successful parsers. Fortunately it is straightforward to make *Parser* also an instance of the *Monad* class.

We proceed as follows. Let us assume that parser p is of type *Par a*, and that q is of type $a \to Par\ b$. Suppose we know the result a of the parser p that serves as the left operand in a monadic composition, then the result of a parser

$$P\ (\lambda k\ input \to p\ (q\ a\ k)\ input)$$

will be of type *Steps a* (*Steps b s*). So all we now have to do is to define a function *getVal*, that makes it possible to

extract the value of type a from this sequence.

```
getVal :: Steps a (Steps b s) → (a, Steps b s)
getVal (Val a  s)  =  (a, s)
getVal (App    s)  =  let (a, s')   = getVal s
                          (f, s'')  = getVal s'
                      in  (f a, s'')
getVal (Shift  v)  =  let (a, r) =  getVal v in (a, Shift  r)
getVal (Done   v)  =  let (a, r) =  getVal v in (a, Done   r)
getVal (Fail)      =  (⊥, Fail)
```

One can see *getVal* as the counterpart of *getProgress*. Whereas the latter manipulates the sequence in such a way that it starts with progress information, it is *getVal* that makes the first value represented in the sequence available, while keeping the progress information. In order to avoid further typing problems we have given *getVal* a type that is somewhat less general than one might expect, in the sense that we require the second parameter of the *Steps* argument to be a *Steps* again. This is not unreasonable; apparently we are not only interested in the embedded value, but we also want to keep the steps, and thus we need a place to return them too.

The instance definition for the monad now becomes:

```
instance Monad (Parser s) where
  return a   = pSucceed a
  P p >>= q  = P  (λk input →
                     let (a, ps_qres) =
                           getVal (p (unP (q a) k) input)
                     in  ps_qres
                  )
```

At first sight this may look confusing, since part of the parsing result is used for the construction of the parser $p\ (unP\ (q\ a)\ k)$ itself. Note however that all the information in the sequence that is needed for this value is produced by the first parser, and thus can be freely used before the second parser is called, and thus before the second parser has to be constructed! Note also that progress information stemming from the use of p is also immediately available in *ps_qres*, even before the value resulting from the call to p has become fully available. Lazy evaluation saves our day again! The value *ps_qres* contains the progress information from p, together with the progress information and result from $q\ a$ (and of course also elements stemming from k). Notice that if the parse p fails then the value a is probably undefined. This does not do any harm because the result of q will never be inspected, being shielded by a *Fail*.

One may wonder what should be the interpretation of a parser such as: *pSym* `'1'` >>= *const* (*return* 1). Should it immediately return the value 1, or should it only do so provided it has first recognized a 1; it is clear from the definitions above that we chosen for the latter interpretation.

Note that we do not recommend using the monadic interface without good reason, since the parsers it constructs do not have online behaviour, for an essential reason — no part of the result of a bind can be produced until at least the first operand has completed parsing. One should only use this interface when there is a good reason to parameterize a parser on a previously parsed value. One good example is a parser that recognizes expressions based on operator priorities: first we recognize the priority definitions, and *bind* them to a function that subsequently parses expressions ([5]). Another nice example is the recognition of XML-like structures:

```
pXML = do t ← pOpenTag
          Tag t <$> pMany pXML <* pCloseTag t

data XML  = Tag t [XML]
pMany p   = (:) <$> p <*> pMany p <|> pSucceed []
```

Finally some might wonder whether the monadic law $p \gg\!= return \equiv p$ holds. In a strict sense it does not since the online behaviour of the left hand side is different from that in the right hand side. They are however equivalent with respect to inspection using *getVal* and *getProgress*.

5. PARSING AMBIGUOUS GRAMMARS

Standard backtracking implementations implicitly produce a search tree of which the leaves represent possible solutions and failures. Such a tree may however contain many identical subtrees, and as such may be more expensive to evaluate than strictly needed. So the question arises whether we can share the computations represented by such subtrees, thus getting a DAG-like structure. In the context of parsing this situation arises if we deal with ambiguous grammars: a specific segment of input may be recognized in many different ways, but the state we are in after all such successful recognitions is the same. Can we can extend our library to recognize such situations, and prevent costly recomputations?

The answer to this question will be partly a *yes*, but before presenting our solution we look into the type of an ambiguous parser. The "list of successes" approach naturally lends itself to dealing with ambiguous grammars, because of the chosen encoding: an empty list denotes failure, a singleton list a single success and a longer list more than one successful parse. So the fact that we return a list value automatically handles the situation in which we deal with ambiguous grammars. Our parsers however always return a single result, and as long as we do not know yet what this single result will be, postpone the point at which they return (part of) any value. So we make essential use of the fact that eventually there will always exactly be one successful path from the root to a leaf in the search tree. So we propose the introduction of a new parser combinator *amb* that indicates that a parser may return more than one result:

```
amb :: Par a → Par [a]
```

The essence of an ambiguous parse is that several parallel parsing "threads" terminate *at the same time*. Since we perform a breadth-first execution of these threads this point is easily recognized, provided we mark the points in the sequence where a parse for the ambiguous nonterminal is completed. In order to be able to do so we introduce one more alternative for the data type *Steps*, called *RS*. This alternative corresponds closely to a shift-reduce state in a bottom-up parser, and contains information about sequences that represent complete parses (reduce), and sequences corresponding to parses that extend beyond this point (shift):

```
data Steps a r  =  ... as before ...
                |  RS [Steps a r] (Steps a r)
```

We first discuss how to extend the function *best*. Since the *RS* essentially is a progress indicator, we decide to deal with it by adding patterns to the definition of *best* directly and not to leave this to *getProgress* somehow. The extended best

```
best :: HasProgress r ⇒
        Steps a r → Steps a r → Steps a r
...
Shift v      `best` Shift      w      = Shift (v `best` w)
RS as     ac `best` RS bs      bc     = RS  (as ++ bs)
                                          (ac `best` bc)
RS as     ac `best` r@(Shift _)       = RS as (ac `best` r)
l@(Shift _)  `best` RS bs      bc     = RS bs (l  `best` bc)
p            `best` q                 = getProgress id p
                                              `best`
                                        getProgress id q
instance HasProgress s ⇒ HasProgress (Steps a s)
  ...
  getProgress f (RS r s)    = RS  (map f  r) (f  s)
```

Figure 5: The extended function *best*

is given in figure 5. The first line represents the already existing case in which both alternatives have not completed yet, and can make progress by shifting a symbol. The next case, with an *RS* pattern at both sides, corresponds to a reduce-reduce "conflict", i.e. a join-point where several ambiguous parses meet again. We do not see the conflict as a conflict however since we want to deal with ambiguous grammars explicitly, and thus just store both results in the first component of the resulting *RS*. Note that it is an essential invariant of our approach that if two *RS* meet each other in a call to *best* they correspond to the same point in the input, since they are both preceded by an equal number of *Shift* steps! The second components of both arguments may contain sequences corresponding to shift actions at this point of the input, so we choose between these shifting sequences with another call to *best*. The next two alternatives correspond to a shift-reduce state, for which we compare the shift side with the other shift sequence incorporated in the *RS* side. Furthermore we extend the definition of *getProgress* for the *RS* case, by storing the f-represented value steps in the components.

It seems that we are done now, but unfortunately we have still some more work to do, since we now have *RS* constructors left in our sequences. Since we allow ambiguous nonterminals to occur inside other ambiguous nonterminals we have to make sure that the *RS* marks corresponding to different non-terminals do not get mixed up. We solve this problem by introducing for each introduction of an *RS* mark a process *unRS* (figure 6), that removes these marks again. This makes everything dealing with ambiguity invisible outside the non-terminal, besides the fact that its type has changed. Now we can give the definition of *amb*:

```
amb (P p)
 = P (λk input
        → unRS id (p (λinp → RS [k inp] Fail) input)
     )
```

The last alternative of the function *unRS* is the most interesting one. The *ls* argument now contains a list of sequences, all starting with *Val* and *App* steps making up the value we are interested in, and a common tail that corresponds to the steps taken afterwards. The *snd* (*head res*) recovers one of these common tails and uses this to build a new sequence that contains at the head a *Val* occurrence with the values

Although we could have made a separate class for overloading the function *unRS* we have decided to extend the class *HasProgress* with an extra function. Notice that when the function *best* is called in the last alternative of *unRS* we know what te type of the "tail" is, but Haskell doesn't; so we had to place a constraint on the function *unRS* for keeping track of this.

```
class HasProgress st where
 getProgress :: (st → Steps x y) → st → Steps x y
 unRS        :: HasProgress y ⇒
                (st → Steps x y) → st → Steps [x] y

instance HasProgress r ⇒ HasProgress (Steps a r) where
 ...
 unRS f (Val a r) =          unRS (f · Val a) r
 unRS f (App r)   =          unRS (f · App )  r
 unRS f (Shift r) = Shift ( unRS  f           r)
 unRS f (Done r)  = error "incorrect program"
 unRS f Fail      = Fail
 unRS f (RS ls r)
    = let  res = map (evalSteps · f) ls
      in   Val (map fst res) (snd (head res))
             `best`
           unRS f r

instance HasRS () where
 ...
 unRS _ _ = ⊥
```

Figure 6: Removing *RS* **marks**

found for the ambiguous parser and the steps produced by the continuation. Of course this new sequence now will have to compete with possible other, still active threads, for this nonterminal. Of course the *RS* mark also has to be removed from this alternative.

One might wonder whether this solution is really online. The answer is that it is not, but also that it cannot be online. The online property essentially depends on the fact that we may produce a result once we know that no other result will be produced, i.e. we have one active thread left. But in the case of an ambiguous grammar we only know that this is the case once we have reached the *RS* mark: in a parser *amb* (p `<*>` q) it may be the case that the ambiguity actually stems from the q, so the fact that we have only one thread while working on the p does not bring us very much.

We finally notice that when constructing the *RS* step in the continuation we expect the continuation k to return some *Steps b r*, so we will have to specialize the type of *Par* a bit:

```
newtype Par a
 = P (∀b r. HasProgress r ⇒
       (String → Steps b r) → String
                            → Steps a (Steps b r)
     )
```

6. SOME OPTIMIZATIONS

Thus far we have not paid attention yet to the efficiency of our algorithms; we have placed emphasis on the presentation instead. in this section we show some improvements that may be necessary for a really useful library.

6.1 Combining steps

There is an easy optimisation which can be quite significant if the function *best* has to compare several steps before being able to decide; i.e. in those cases where we need a significant look-ahead.

When *getProgress* traverses the progress-free part of a parser, is accumulates the information found there, and reinserts this after the first encountered step. If *getProgress* is subsequently applied again to the constructed result, it will traverse this progress-free prefix again, which is clearly wasteful. The waste can be avoided by introducing a new alternative of *Steps*, representing a suspended *getProgress* computation:

$$\begin{array}{l}\textbf{data}\ \mathit{Steps}\ a\ s\\ \quad = \ \ldots\ \mathit{as\ before}\ldots\\ \quad \mid\ \exists b\ x.\ \mathit{Mix}\ (\mathit{Steps}\ b\ x \to \mathit{Steps}\ a\ s)\ (\mathit{Steps}\ b\ x)\end{array}$$

Mix f s represents the sequence f s, with an invariant that f only adds progress-free steps. Now we can modify *getProgress* to construct a suspension *Mix* f s a soon as it encounters its first progress step, and to continue directly from that point on if applied to the same sequence again. The new cases are:

$$\begin{array}{lcl}\mathit{getProgress}\ f\ (\mathit{Done}\ \ p) & = & \mathit{Done}\ (\mathit{Mix}\ f\ p)\\ \mathit{getProgress}\ f\ (\mathit{Shift}\ \ s) & = & \mathit{Shift}\ (\mathit{Mix}\ f\ s)\\ \mathit{getProgress}\ f\ (\mathit{Mix}\ g\ \ s) & = & \mathit{getProgress}\ (f \cdot g)\ s\end{array}$$

Other functions on *Steps* just treat *Mix* f s as f s.

6.2 Special cases

The sequences constructed contain many frequently occurring subsequences that, if they are represented by special alternatives, can reduce the length of the sequence considerably. We mention a few that may be introduced, but we will not discuss the extensions to the functions involved since they are all straightforward.

To start with we observe that quite often a *Shift* is immediately followed by a *Val*, so it makes good sense to have a special kind of step *ShiftVal*, that acts both as a shift and as a *Val*.

As we can see from the examples given there are many occurrences of `<$>` and `<$` in our parsers. For these we may introduce a special step kind:

$$\begin{array}{l}\textbf{data}\ \mathit{Steps}\ a\ r\ =\ \ldots\\ \qquad\qquad\qquad \mid\quad \mathit{ShiftVal}\ a\ r\\ \qquad\qquad\qquad \mid\ \exists b.\ \mathit{AppVal}\ (b \to a)\ (\mathit{Steps}\ b\ r)\\ f\ \texttt{<\$>}\ q\ = (\mathit{AppVal}\ \ f\ \cdot) \cdot q\\ f\ \ \texttt{<\$}\ \ q\ = (\mathit{AppVal}\ (\mathit{const}\ f) \cdot) \cdot q\end{array}$$

Similarly, we notice that in many cases we are not interested in the result of the parser, and just want to recognize something. In such cases it is wasteful to insert the values into the sequence, together with a function which removes them from the result. An efficient solution is to tuple every parser (constructor P) with a recognizer (constructor R) that recognizes the same sequence of input tokens but does not return a result; so instead of constructing parsers out of parsers and recognizers out of recognizers we now construct pairs of a parser and a recognizer out of such pairs.

$$\begin{array}{l}\textbf{type}\ \mathit{PR}\ a = (\mathit{Par}\ a,\ \mathit{Rec})\\ \begin{array}{lllcl}f\ \texttt{<\$>} & (P\ p, & R\ r) & = & (P\ ((\mathit{AppVal}\ f\cdot)\ \cdot p)\\ & & & & \ ,R\ r\\ & & & &)\\ f\ \ \texttt{<\$} & (P\ p, & R\ r) & = & (P\ ((\mathit{Val}\ \ f\cdot)\ \ \cdot r)\\ & & & & \ ,R\ r\\ & & & &)\\ (P\ pp, R\ pr)\ \texttt{<*>} & (P\ qp, & R\ qr) & = & (P\ ((\mathit{App}\cdot)\cdot\ pp\ \ \cdot qp)\\ & & & & \ ,R\ (\ \ \ \ \ pr\ \ \cdot qr)\\ & & & &)\\ (P\ pp, R\ pr)\ \texttt{<*} & (P\ qp, & R\ qr) & = & (P\ (\ \ \ \ \ pp\ \ \cdot qr)\\ & & & & \ ,R\ (\ \ \ \ \ pr\ \ \cdot qr)\\ & & & &)\end{array}\end{array}$$

Lazy evaluation will make that only parsers that are actually used are constructed.

7. PRACTICAL POLISH

The techniques described in this paper have been included, although in a heavily optimized way, in a parsing library, together with many other libraries and tools[4]. This combinator library ([4]) produces parsers that parse at about half the speed of off-line generated parsers, while also preparing for and performing full error repair and error reporting.

This repair process was addressed by Swierstra and Azero ([5, 4]), who describe a parsing strategy in which an error repairing parser returns a sequence of steps that represent a trace of progress of the parsing process *and* the taken repair actions. The corresponding parsing result was built up in an accumulating parameter, and is appended to the parsing trace when a successful parse has been found. Hence that solution did not have online behaviour. The basic idea of the error repair is that a parser doesn't just fail when a symbol cannot be recognized, it proceeds in parallel along two alternative routes: insert the symbol expected, and delete the current input symbol and try again. Once we associate a specific cost with each insertion and deletion we can use a more general version of our function *best* to select between all the different possible corrections, according to whatever strategy one might wish to express. One of the nice aspects of the introduction of the data type *Steps* is that nothing prevents us from incorporating even more information in the sequence by adding extra alternatives, provided they once again form an 'orthogonal' subsequence whose elements may be freely shifted back and forth in the main sequence, provided their mutual order does not change. We use this for getting tracing information out of the parsing process and for constructing error messages.

8. OTHER APPLICATIONS

We have derived a small parsing library using a number of orthogonal concepts. Looking back at the code constructed we can see three different elements:

1. a data type *Progress* with a function *best*
2. a data type *Steps*, with a class *HasProgress* containing a function *getProgress* (and possibly *unRS*)
3. a set of parser combinators generating sequences consisting of the first two elements

[4]http://cvs.cs.uu.nl/cgi-bin/cvsweb.cgi/uust/INSTALL.html

We notice that the first two elements are of independent interest, and that there is nothing in them that is particularly connected to parsing. Essentially elements of the data type *Progress* represent search trees, with *Shift* applications forming the edges, *Done* and *Fail* as leaves, and calls to *best* for forming branching nodes and representing the breadth-first searching process. A path in such a tree may furthermore contain a value, represented as a sequence of fragments interleaved with *Shift*'s.

The single role of the parser combinators is to generate this branching tree, but we may easily envision other applications that generate similar structures.

A second observation is that the trick we applied to the data type *Expr a* in converting it to a data type *Polish a r* can be applied to all data types. Suppose we need a sequential representation of a data type T. Now provide all data types reachable from T with an extra parameter and rewrite each right hand side of an alternative in a "continuation" passing style. As an example consider the following two definitions for *RoseTree*: the standard one and its continuation style counterpart (the $'$-ed identifiers):

```
data RoseTree a      = Node a [RoseTree a]
                     | Leaf

type IntRoseTree     = RoseTree Int

data RoseTree' a' r  = Node' (a' (List (Rosetree' a)) r))
                     | Leaf' r
data List' f' r      = Cons' (f' (List' f' r))
                     | Nil' r
data Int' r          = Int' Int r

type IntRoseTree' = RoseTree' Int'
```

In our example we have used the type *Polish* because parsers are polymorphic in the result they return, and we should thus be able to represent any kind of value. The technique may be useful in many other contexts.

9. WNIOSKI (CONCLUSIONS)

We have shown that we *can* represent Polish expressions in Haskell in such a way that progress information such as recognizer steps can be merged with the Polish, and a lazy interpreter can be defined, permitting applications that produce Polish to work in an "online" manner. Applications that output Polish can be run "in parallel", by synchronizing their progress, and this in turn permits a breadth-first exploration of a search space. The Polish idea underlies a high performance parsing library, which demonstrates its practicality, and we hope it may find other applications too.

10. ACKNOWLEDGEMENTS

We want to thank Arthur Baars, Atze Dijkstra and Daan Leijen for helpful comments on earlier versions of the paper, and we thank Andres Löh for his help with the lhs2TEX system.

11. REFERENCES

[1] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 1–52. Springer-Verlag, Berlin, 1995.

[2] G. Hutton and E. Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[3] D. J. P. Leijen and H. J. M. Meijer. Parsec: Direct style monadic parser combinators for the real world. UU-CS 2001-35, Department of Computer Science, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands, 2001.

[4] S. D. Swierstra. Combinator parsers: From toys to tools. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, 2001.

[5] S. D. Swierstra and P. R. Azero Alcocer. Fast, error correcting parser combinators: a short tutorial. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99 Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics*, volume 1725 of *LNCS*, pages 111–129, November 1999.

[6] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

[7] E. Visser. Scannerless generalized-lr parsing. P 9707, Programming Research Group, University of Amsterdam, July 1997.

[8] P. L. Wadler. How to replace failure by a list of successes. In J. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 113–128. Springer-Verlag, 1985.